Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Jukka Rasi

Blockchain distributed DNS without trust:
Publishing IOT device addresses and verifying data

Master's Thesis
Espoo, October 30, 2017

Supervisor: Professor of Practice Kary Främling
Advisor: Professor of Practice Kary Främling

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Jukka Rasi		
Title:	Blockchain distributed DNS without trust: Publishing IOT device addresses and verifying data		
Date:	October 30, 2017	**Pages:**	45
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Professor of Practice Kary Främling		
Advisor:	Professor of Practice Kary Främling		

Blockchain enabled distributed DNS makes possible to have a trustless system, where no participant needs to be trusted. Blockstack is such a distributed DNS that is built on top of Bitcoin's blockchain. In this thesis I will extend this trustless feature to data sharing from an IOT device, by creating a proof of concept implementation. Cryptographically linking parts together, the trustless feature of the underlying blockchain can be preserved from the blockchain to the shared data from the device.

Keywords:	bitcoin,blockchain,IOT,blockstack,cryptography
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Master's Programme in Computer, Communication and Information Sciences

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Jukka Rasi		
Työn nimi:	Lohkoketjun hajautettu DNS ilman luottamusta: IOT laitteen osoitteen julkaisu ja datan verifiointi		
Päiväys:	30. lokakuuta 2017	**Sivumäärä:**	45
Pääaine:	Computer Science	**Koodi:**	SCI3042
Valvoja:	Professori (Professor of Practice) Kary Främling		
Ohjaaja:	Professori (Professor of Practice) Kary Främling		

Lohkoketjun päälle rakennettu hajautettu DNS mahdollistaa järjestelmän jossa ei tarvitse luottaa muihin osapuoliin. Blockstack on tällainen hajautettu DNS, joka on rakennettu Bitcoinin lohkoketjun päälle. Tässä työssä laajennan tämän luottamattomuus ominaisuuden IOT laitteen datan jakamiseen demo-ohjelman muodossa. Kryptograafisesti linkittämällä eri osat toisiinsa, voidaan perustalla olevan lohkoketjun luottamattomuus ominaisuus laajentaa myös IOT laitteen jakamaan dataan asti.

Asiasanat:	bitcoin,lohkoketju,IOT,blockstack,kryptografia
Kieli:	Englanti

Acknowledgements

Thanks to Professor of Practice Kary Främling for giving me the opportunity to work on this thesis as a Research Assistant. Cheers to friends and family for not letting me give up!

Espoo, October 30, 2017

Jukka Rasi

Abbreviations and Acronyms

IOT	Internet Of Things
MitM	Man-in-the-Middle attack
PGP	Pretty Good Privacy
PKI	Public Key Infrastructure
CA	Certificate Authority
SSL	Secure Sockets Layer
TLS	Transport Layer Security
DSA	Digital Signature Algorithm
ECC	Elliptic Curve Cryptography
UTXO	Unspent Transaction Outputs Cache
DoS	Denial of Service
DNS	Domain Name System
DHT	Distributed Hash Table
DL	Discrete Logarithm
EC	Elliptic Curve
RSA	Rivest, Shamir, and Adleman
RAM	Random Access Memory
IHV	Initial Hash Value
TSA	Time Stamping Authority
GUI	Graphical User Interface
SegWit	Segregated Witness
ODF	Open Data Format
OMI	Open Messaging Interface
API	Application Programming Interface
XML	eXtensible Markup Language
JSON	JavaScript Object Notation
REST	Representational State Transfer
DOM	Document Object Model

Contents

Chapter 1

Introduction

The purpose of this thesis is to explore if Blockstack's distributed DNS and PKI can be used to share IOT node addresses and data without a central authority in a trustless manner. Blockstack is a distributed DNS and PKI that is build on top of Bitcoin's blockchain. The underlying blockchain provides the trustless environment.

Three goals are to be tested:

- **Finding nodes**. The Blockstack's DNS is used to find the IOT node addresses based on a unique identifier known to the user in a verifiable way.
- **Verify node data**. The data shared by the nodes should be linked by public key cryptography to the unique identifier found from Blockstack's DNS. User can verify that the data shared by the IOT node is published by the owner of the unique identifier.
- **Trustless environment**. The trustless feature of the blockchain is shared by the finding of nodes and data verification. Users don't need to rely on any central authority.

Finding IOT node addresses

IOT nodes have a address which can be found from Blockstack's DNS using a unique identifier. This address is cryptographically linked to the unique identifier, thus the searcher can always verify that the address is provided by the owner of the unique identifier. The unique identifier has been provided to the searcher by the owner of the id in verifiable way, as in person.

Verifying node data

The nodes will also have generated their own public and private key combo and the public key will be shared through Blockstack's DNS together with the node address. Data will be signed with the nodes private key and this way the validity of the data can be checked against the shared public key.

Trustless environment

Conventional systems like DNS and SSL rely on trusting some counter-party. In DNS most of the trust relies on the root DNS servers. In addition to trust there are reliability issues: the root servers are an easy target for denial of service (DOS) attacks. SSL hierarchy of trust also has the same issues with trust and reliability. Users need to trust the central authority (CA) to verify sites and to remain neutral while doing so. Central authorities are single points of failure, which are attractive targets for hackers.

Instead the Blockstack's DNS and PKI system runs on top of Bitcoin's blockchain, which makes the DNS and PKI system as secure as the underlying blockchain. Bitcoin's blockchain represents a public ledger, which can not be modified maliciously without controlling a majority of the computing power in the network.

In this thesis I will explore if it is possible to extend this trustless feature to the data shared by IOT Nodes.

Thesis Structure

The next chapter *Background* will go through some basic background information needed to understand how this trustless feature emerges from blockchains. Next in *Setup* I will briefly introduce the programs and environment needed to develop and run the implementation. In *Methods* the plan is explained in a higher level before going through it in more detail in *Implementation*. *Evaluation* discusses how the plan worked and explores possible encountered problems. Chapter *Conclusions* concludes the thesis with final remarks.

Chapter 2

Background

To understand how trust works in a distributed DNS system like Blockstack we need to explain some of the fundamentals of Bitcoin and cryptography. We start by looking at the original paper by Satoshi Nakamato which introduced the electronic currency Bitcoin. After that the fundamentals of public key cryptography are explored, including hashing and digital signatures. Then the concepts of time-stamping, hashing and proof-of-work are linked together to form the basis of the distributed system behind Bitcoin that provides the trustless environment. Next some of the technical details of transactions are explained more closely, as the Blockstack implementation is build on top of them. Last the Blockstack architecture with the virtualchain is explained. This finally shows how the trustless blockchain network enables the distributed DNS to have the same trustless feature embedded in it.

Bitcoin and Blockchain

Bitcoin was introduced by a pseudonym Satoshi Nakamoto in the paper "Bitcoin: A Peer-to-Peer Electronic Cash System" in 2008. In the paper Nakamoto criticizes the current electronic payment systems that rely heavily on trusted third parties. Mediation costs increase the transaction costs and limit the minimum transaction size, making small payments an impossibility. The possibility of reversals also increase the amount of information merchants need from customers to be able to trust them. The costs and uncertainties can be avoided by using physical currency, but before Bitcoin there was no way to make payments over the Internet without a trusted third party. [17]

Previous electronic payment systems also have the problem of double spend.

doublespend

The coins used in a electronic payment system are usually in some way copyable, which leads to the problem, that when you receive a coin, you can not be certain that the same coin has not been used in a previous transaction. As a solution to the double spend problem, the issuer of the coin has usually had to have a database listing all of the transactions, to make sure that no coin has been spent multiple times. This solution leads to a new problem. The entire money system now depends on the company issuing the money and all transactions need to go through them, just like with a bank. [17]

Nakamoto introduces a solution to these problems, that bases on cryptographic proof, instead of trust. A electronic payment system that enables transactions between parties, without a trusted third party. The technology behind this electronic payment system (Bitcoin) is called the blockchain, which ingeniously combines some well known cryptographic methods, such as private and public key cryptography and hashing. In the next chapters I will go through the building blocks of blockchain. To understand the inner workings of Bitcoin and blockchain we should first start by looking in to public key cryptography.

Public Key Crypthography

Bitcoin and blockchain rely heavily on public key cryptography to work, so in this chapter we look more closely in to public key cryptography.

Cryptography is a method to communicate securely in the presence of an attacker who can listen in or control the communication channel. Symmetric cryptography is concerned with encryption: how to scramble a message so that only the receiver is able to decipher it. Kerckhoffs's Principle states that the encryption key should be kept private, but the algorithm should be public. This way the algorithm can be publicly tested against vulnerabilities the original designer might have not thought of. [16]

Public key cryptography was developed, because of a key vulnerability in symmetric encryption: key distribution. When two people are communicating with symmetric encryption, they need to share the same encryption key beforehand through a secure channel. This is not always possible. [16]

If two participants (Alice and Bob) want to communicate using public key cryptography, they start by generating a public-private keypair for themselves. Public-private keys are mathematically linked in a way that if a message is encrypted using one of them, it can only be opened by using the other key. For Alice and Bob to communicate, they can send messages

by encrypting them with the other ones public key. Now only the owner of the public key has the corresponding private key to open the encryption and thus read the message. This however still leaves the problem of how to communicate the public keys to each other and opens the possibility for a *Man-in-the-Middle attack* (MitM). [16]

The Man-in-the-Middle attack is an attack where the attacker intercepts messages between participants on an insecure channel. The attacker intercepts the public key sent by Alice to Bob and instead sends his own key. The attacker can now eavesdrop on the messages or even alter them, without the participants ever knowing. There is a couple of ways to handle the MitM attack. [16]

- The participants could share their public keys by another, more secure channel, like in person, but this is again a similar problem as with key distribution. However this **public key distribution** problem is easier, as it only needs authentication, not authentication and privacy.

- **Web of trust**. A method used in *Pretty Good Privacy* (PGP) and *GNU Privacy Guard* (GPG). Users sign the public keys of other users they know and trust. Public keys are kept in a **keyring**. If the user needs to send a message to someone he does not know (public key not in keyring), the user can request the public key from someone he trusts (public key in keyring) to sign and send it.

- **Public Key Infrastructure (PKI)**. Traditional PKI assumes a central authority called the **Certificate Authority (CA)**. Everyone has the public key of the CA and trusts it. User shares his public key with the CA and the CA checks the users identity and then signs the public key. Now when the user wants to communicate with someone else, he sends the CA signed public key. The other user can now check that the public key is properly signed by the CA, i.e. that the certificate is valid. This is essentially how Secure Sockets Layer (SSL) and Transport Layer Security (TLS) work. SSL and TLS are the most widely used protocols for secure communication on the internet.

[16]

A concept that is closely related to public key cryptography is digital signatures. In Bitcoin and blockchain the transactions are signed to prove they are made by the owner of the funds that are being transferred. In the next section we will go through how digital signatures work.

Digital Signatures

The goal of digital signatures is to ensure that the message was generated by the signer, has not been tampered and to make sure that the signer is not able to deny having signed it. As digital signatures are signed with a private key, which is linked to a public key, the signature can always be proven to have come from the owner of the public key. In comparison, when using symmetric algorithms, proving that the message came from the sender is not possible, as both the sender and receiver share the same key used in the encryption. [16]

In the Bitcoin protocol, digital signatures are used to sign transactions. A bitcoin address is a public key and if the owner wants to make a transaction he needs to sign the transaction with the corresponding private key. The messages signed with digital signatures can be of an arbitrary length. Public key cryptography algorithms are quite slow, so long messages would take a long time to process. Because of this, hashes are usually signed, instead of the whole message. **Hash function** is a algorithm that takes arbitrary length data as input and outputs a fixed length bit-string, called the **hash value**. Hash functions produce outputs that are always the same for the same input, but small changes in the input produce big changes in the output. [16] Hashes are explained more closely later when discussing the concept of proof-of-work in Bitcoin and blockchain.

Digital signature protocol is a combination of a public-key algorithm and digital signature scheme.

Main *public key families* used:

- **Integer factorization**. Based on factoring large integers. For example RSA (Rivest, Shamir, Adleman). [18]
- **Discrete logarithm (DL)**. Based on the difficulty of the discrete logarithm problem on finite cyclic groups. [15]
- **Elliptic curve (EC)**. Based on hard task of computing the generalized logarithm problem on an elliptic curve.

[16]

Main *digital signature schemes*.

- **RSA**. RSA signature scheme is based on the RSA algorithm. Most widely used digital signature scheme.
- **Schnorr signature**. Simplest scheme that can be used with both discrete logarithm and elliptic curve algorithms. Small computational times for signing and verification and smaller signatures.

- **Elgamal signature**. Works for both the discrete logarithm and elliptic curve algorithms. Not widely used in practice as it needs more computation and the signature is bigger than in other schemes like Schnorr or DSA.

- **Digital Signature Algorithm (DSA)**. Widely used. Patent covering it was made worldwide royalty-free.

[16]

Bitcoin protocol uses *elliptic curve cryptography* (ECC) with the DSA signature scheme. When RSA [18] is based on factoring large integers, ECC is instead based on an elliptic curve consisting of large set of successive points. Private and public keys are points on this curve. [16]

The next chapter discusses how Bitcoin addresses are constructed from these ECC public keys.

Bitcoin addresses

A Bitcoin address is a hash of the ECC public key. OpenSSL is used to perform elliptic curve cryptography. 65 bytes are used in OpenSSL to represent a elliptic curve. First byte is used to store the type of the point: 0x04 uncompressed and 0x02 or 0x03 for compressed. If the omitted coordinate (y-coordinate) was even, then 0x02 and if it was odd, 0x03 is used. After the first byte, in the uncompressed representation comes the x and y coordinates and in compressed representation, only the x coordinate. Bitcoin uses 256 bit elliptic curves, so each coordinate takes 256 bits = 32 bytes of space. Uncompressed point then takes a total of 65 bytes (1 for the type and 32x 2 for the coordinates). Next the OpenSSL representation of the elliptic curve point is hashed using SHA256 and the RIPEMD160. SHA256 produces a hash of 256 bits = 32 bytes. RIPEMD160 produces a hash of 160 bits = 20 bytes. This second hashing was chosen by Satoshi to reduce the size of the address, thus making transactions smaller.

For preventing typing errors, a checksum is also calculated from the hash. The 20 byte RIPEMD160 hash is run through a $SHA256^2$ and again through a RIPEMD160 hash and then the first 4 bytes are kept as a checksum. The byte string starts with a byte indicating the *type of the address*, next the 20 bytes resulted from the RIPEMD160 hash and last the 4 byte checksum. The type of the address can be:

- **0s** (decimal) for a **public key address**. The encoded address would start with *1*.

- **5s** (decimal) for a **script address**. The encoded address will start with a *3*.
- **111s** (decimal) for a testnet public key. Encoded address starts with an *m* or *n*.
- **196s** (decimal) for testnet script address. Encoded address starts with *2*.

In the last step the byte string is encoded with Base58, resulting in a Bitcoin address that is 27 to 34 characters long, having characters A-Z, a-z, 0-9, excluding 0,O,I and l. [16]

Bitcoin transactions are made to and from these addresses. Next we will go through how transactions work.

Transactions

Bitcoins are not stored in users computers, but instead they are entries in a distributed database, called the blockchain. The blockchain does not store accounts and balances, but transactions. Transactions are a combination of a list of **transaction inputs (TxIn)** and a list of **transaction outputs (TxOut)**. Each transaction ouput has an amount and the recipient address. Transaction input has a reference to a previous transaction output and a signature that proves that the funds in the previous transaction output can be spent. The signature must be signed by the private key linked to the public key in the bitcoin address. If the signature does not match, the network will drop the invalid transaction. A transaction holds several TxIns and TxOuts and distributes funds from input to outputs. The outputs must be unspent or the transaction is invalid. For the transaction to be valid, the sum of inputs must be greater than or equal to the sum of outputs. The difference between inputs and outputs is called the **transaction fee** and it is payed to the miner of the block that holds the transaction. Outputs in the blockchain can only be spent once and they must be spent fully. If the sender does not want to pay all of the remainder as a transaction fee, he can include a **change address** as an additional output to the transaction. The origin of funds can be used, but to increase privacy, it is recommended to create a new address after every transaction. [16]

When a transaction is sent to the network, the first node receiving the transaction will check if it is valid and then passes it on to other nodes. Steps to check validity:

- Check that the previous outputs exist and have not been spent.

- Check that the sum of inputs is greater than or equal to the sum of outputs. Difference left as payment for the miner.
- Check the signature validity for each of the inputs.

[16]

To make the checking of outputs faster, Satoshi forced that the output should be spent fully. This enables bitcoin to have a **unspent transaction outputs cache**(UTXO), which makes checking outputs faster. All of the transactions inputs should be listed in the UTXO and if some are missing, the transaction is invalid and can be discarded. UTXO uses a lot less disk space than the whole blockchain and this allows nodes to keep it in RAM, making transaction checking faster. [16]

Transaction outputs can be more than just a bitcoin address. This will be explored in the next chapter with transaction scripts.

Transaction scripts and types of transactions

The Bitcoin protocol allows the transaction output to be more than just a bitcoin address. Transaction output creates a mathematical puzzle that must be solved in order to spent the output. Script that creates the puzzle is called <**scriptPubKey**> and the script that unlocks the funds is called <**scriptSig**>. A node that is checking the validity of a transaction, concatenates the <scriptPubkey> with <scriptSig> and if the final result is true, the transaction is deemed valid. The scripting language is *stack-based*. Commands called **opcodes** place data on the stack, or operate on the data in the stack. Operations can only work on the data that is at the top of the stack. The scripting language is not **Turing-complete** to avoid certain attacks on the network. With a Turing-complete language, an attacker could create <scriptPubKey> that never finishes, thus stalling the nodes running the script and possibly crash the whole network. [16] Some alternatives to Bitcoin, such as Ethereum implement a full Turing-complete transaction system [8].

There are several different types of transactions.

Pay-to-address is the most common type of transaction. In it the previous transactions TxOut has a <scriptPubKey> with the following commands in order:

- OP_DUP
- OP_HASH160
- <hashPubKeyHex>

- OP_EQUALVERIFY
- OP_CHECKSIG

And the current transactions TxIn has a <scriptSig> with:

- <sig>
- <pubKey>

A example execution of the commands and data are shown in table 2.1.

Stack	Command
	<sig> <pubKey>
<sig> <pubKey>	OP_DUP
<sig> <pubKey> <pubKey>	OP_HASH160
<sig> <pubKey> <pubKeyHash>	<hashPubKeyHex>
<sig> <pubKey> <pubKeyHash> <hashPbKeyHex>	OP_EQUALVERIFY
<sig> <pubKey>	OP_CHECKSIG
1	

Table 2.1: Bitcoin Pay-to-address transaction

The stack is LIFO (last in first out), so the data and and commands that are pushed first to the stack, are used last [16]. There are several different transaction types defined in Bitcoin, which are listed in the next chapter.

6.7 Standard Transactions

Bitcoin defines six transaction types:

- **TX_PUBKEY**, pay-to-public-key. The <scriptPubKey> for this is [OP_PUBKEY OP_CHECKSIG].
- **TX_PUBKEYHASH**, pay-to-address. The <scriptPubKey> is [OP_DUP OP_HASH160 OP_PUBKEYHASH OP_EQUALVERIFY OP_CHECKSIG].
- **TX_SCRIPTHASH**, pay-to-script-hash (P2SH). The <scriptPubKey> is [OP_HASH160 <20-byte-hash> OP_EQUAL].
- **TX_MULTISIG**, multisignature transaction. The <scriptPubKey> is [m sig1 ... sign n OP_CHECKMULTISIG].
- **TX_NULL_DATA**, also known as OP_RETURN transactions. The <scriptPubKey> is [OP_RETURN <data>].

- **TX_NONSTANDARD** if it is anything else.

The first five are called the **standard transactions** as they are the only transactions forwarded or mined by nodes implementing the reference Bitcoin software. Non-standard transaction can only be included by a miner of a block [16]. For the purpose of this thesis, the **OP_RETURN or TX_NULL_DATA** is especially interesting, as it is used by the Blockstack implementation to create the distributed DNS. This will be explored later on when discussing Blockstack's architecture.

When the transactions are linked together one after another, they form the actual **blockchain**. In the next chapter we will look more closely to the building blocks of the blockchain: hash functions, time-stamps and proof-of-work.

The blockchain

The blockchain secures the distributed network with **proof-of-work**. Computational power is needed to do the proof-of-work and thus create the transactions blocks. For an attacker to successfully change a transaction in a block, he would need to have enough computing power to do the proof-of-work for the block that has the transaction, for all the blocks that has been created after that, and keep adding blocks faster than the rest of the network combined[16].

Next we will look in to some of the technologies behind the blockchain that makes this kind of an protection possible.

Hash functions

Transaction blocks are linked together by their hashes. Lets look more closely how a hash function works.

Hash function is an algorithm that takes arbitrary length data as input and outputs a fixed length bit-string, called the **hash value**. Hash functions produce outputs that are always the same for the same inputs, but small changes in the input produce big changes in the output. Hash function should distribute input values to hash values proportionally, so that every hash value maps to roughly same number of input values. [16]

Proof-of-work in Bitcoin uses **cryptographic hash functions**. These cryptographic hash functions have a few extra requirements:

- **One-wayness**. It is computationally infeasible to find the input value from the hash value.

- **Weak collision resistance**. It is computationally infeasible to find another input with the same hash value.
- **Strong collision resistance**. It is computationally infeasible to find two inputs that result in the same hash value.

Bitcoin uses $SHA256^2$ (SHA256 two times) for its proof-of-work function. SHA256 fulfills the one-wayness requirement. There is no known algorithm that can recover the message in polynomial time compared to the size of the message. In practice, the only way to break the hash, is to brute-force it, and this takes exponential time. [16] This feature is used in the proof-of-work, which we will go through later.

SHA256 uses the **Merkle-Damgård construction** as a **compression function**. The compression function outputs same length messages as the input, but the bits are scrambled in a deterministic, but complicated way. The message is first broken to blocks of 256 bits. The end of the message is padded with zeros and the length of the message. Now the 256 bit blocks are ran through the compression function, which takes in a *Initial Hash Value*(IHV) and the message which is in our case is one of the 256 bit blocks. The IHV is always the previous blocks hash and the result of the last blocks compression is the SHA256 hash value of the whole message [16]. With $SHA256^2$ the results of the first SHA256 are ran through again and the end result is $SHA256^2$ which is used in Bitcoin.

In the context of the blockchain, hashing is used to calculate a hash of the whole block of transactions, with a few additions that enable it to have the features of linking blocks and proof-of-work. Before explaining proof-of-work in more details, lets first look in to how time-stamping works in the blockchain.

7.2 Time-stamp

Digital time-stamps are used to prove that some information existed at a particular time. Usually a hash of the data is used in the digital time-stamp, instead of the whole data. This is more secure (data and time-stamping medium can be kept separate) and uses less space (hash is usually smaller). [16]

One way to secure a digital time-stamp would be to sent a trusted party a copy of the information, which would be stored together with the time of reception. The trusted party could however lose the database or get compromised. One could also use a **Time-stamping authority** (TSA) to sign the data and time-stamp with TSA's own private key and then store the signature themselves. This however relies on trusting the TSA to be honest. [16]

A third way to secure a digital time-stamp would be to publish the hash in a public place. This could be, for example, a newspaper. An attacker would need to crack the published hash, or subvert all the newspapers with the published hash. TSA could use this to publish a hash of the signature of a time-stamp. However, publishing every time-stamp separately would be costly. The costs could be brought down by combining the hashes of multiple time-stamps before publishing. Multiple hashes of data to be time-stamped can be combined with **Merkle trees**. [16]

To make the digital time-stamps even more secure, **linked time-stamps** can be used. The next data hash to be published would be linked to the previously published hash. Now an attacker needs to crack two hashes to be able to make changes. Linking every hash to a previous one, makes cracking exponentially harder, as more links are added. [16]

A TSA would work by first collecting data from clients to be time-stamped for some time period. When the period is over, the collected data is hashed together using a Merkle tree, the resulting hash is hashed together with the final hash of the previous time period and then the final hash would be published. This scheme could be used to secure transactions in a digital currency, but it would rely on trusting the central counter party running the TSA and would also need some kind of an public medium to publish the hashes. In Bitcoin, proof-of-work is the last addition to this scheme, that allows a distributed system without the need for trust. [16]

Lets next explain what is proof-of-work and how it relates to hashes.

Proof-of-work

Digital services are subject to several kinds of attacks, such as denial of service (DoS). One defense against attacks is to require clients to prove that some kind of work has been done (Proof-of-work). The proof-of-work could be for example some problem needing user input, like CAPTCHA, but in Bitcoins case, a computationally-hard problem is used. The problem should be hard to solve, but easy (fast) to verify. Proof-of-work systems can follow two different protocols:

- **Challenge-Response**. Client requests access to a service and the service poses a proof-of-work challenge to the client. Server grants access to the client only if the challenge has been solved correctly. This is for example the model used with CAPTCHA.
- **Solution-Verification**. This is a asynchronous protocol in which ongoing communication between client and server is not needed. Solution and verification can be done at different times. Client creates a

proof-of-work problem according to a predetermined and agreed upon algorithm and then solves it. Then the client sends the solution to the server for verification.

Bitcoin requires proof-of-work to be performed on transaction blocks before they can be added to the blockchain. **Partial hash inversion** is used as the proof-of-work function. A hash of a block of transactions must match a pattern in which the hash starts with a certain number of 0 bits. This is called hash inversion as the proof-of-work has to invert a certain pattern in part of the hash. To solve the partial hash such a *nonce* has to be found that when applied to the message the resulting hash will start with the required amount of 0 bits. It is computationally costly to solve the partial hash, as many nonces have to be tried before finding the solution. It is however easy to verify as only one hash calculation is needed. The difficulty of the problem can easily be adjusted by changing the required amount of 0 bits. Bitcoin borrowed the idea of partial hash inversion from **Hashcash** that was introduced by Adam Back in 1997 [13]. [16]

Clients that are doing this proof-of-work are called **miners**. Miners use electricity to compete on finding the next proof-of-work hash. To encourage this behaviour, miner of a block receives bitcoins which consists of a **block reward** and **transaction fees**. The block reward is the mechanism by which new bitcoins are created. It started as 50BTC and is halved roughly every 4 years. Currently the reward is 12,5BTC and the bitcoin supply grows asymptotically approaching 21 million, but never reaching it exactly. The 21 million cap is going to be reached in practice around year 2140. [2]

Next lets summarize the blockchain technology before introducing the Blockstack's distributed DNS and PKI.

Blockchain summary

Blockchain technology was invented with the cryptocurrency Bitcoin. It consists of a network of nodes running the Bitcoin software using electricity to compete on finding transaction blocks and receiving block rewards and transaction fees. This competition based on cryptographic proof-of-work creates the trustless environment for the blockchain. This means that the only thing the participants have to trust, are the algorithms behind Bitcoin, which can only be changed by the majority vote of the network (hard forks). No third party is needed to verify the transactions or handle the ledger.

The blockchain provides a immalleable public ledger that has the full history of transactions recorded. It consists of blocks of transactions linked by

previous blocks hashes, creating a chain of blocks, *blockchain*. Units of currency are stored as transactions instead of amounts in accounts. Transaction addresses are ECC public keys and transactions are made by using public key cryptography to sign the transactions. Miners collect the transactions to blocks and try to solve the problem of partial hash inversion to create the next block in the chain and receive the rewards.

While the Bitcoin blockchain is currently mostly used only as a store of value, the trustless environment and public ledger enable a variety of apps to be built on top of it. **Blockstack** is one of these apps, which works as a distributed DNS and PKI. The next chapter will go through Blockstack's architecture and technology and how we can use it with the IOT nodes.

Blockstack

Blockstack is the technology used to create a distributed DNS and PKI on top of the Bitcoin's blockchain. In this chapter we will first go through Blockstack's history and the reasons for building it, and then take a look at the architecture behind the technology.

Blockstack's history

Before Blockstack was created to run on top of another blockchain, it ran on it's own blockchain called the Namecoin. It had over 33 000 registered entries and 200 000 transactions, before migrating to Bitcoins blockchain as Blockstack. [12]

Running the Namecoin blockchain revealed a number of security and reliability concerns. A single miner was able to control over 51% of the mining power, thus having the power to launch a successful 51% attack on the network. There were also some problems related to broadcasting transactions reliably over the network. Reliability of the network depends on how actively a blockchain network is used and thus on how much financial incentives there are to add more computing power. For security and reliability reasons blockchain-based services should use the largest and most secure blockchain, which at the moment is the Bitcoin blockchain. The most used blockchain also usually has the most active code base, thus making bug fixing faster. Introducing new features to old blockchains is hard, as they are usually consensus-breaking and starting new blockchains is less secure. That is why Namecoin migrated to Blockstack to overcome these tradeoffs with a virtualchain build on top of another (currently Bitcoin) blockchain. While working with Namecoin, the developers learned five lessons:

1. There is a tradeoff between blockchain security and introducing new functionality to blockchains.
2. There is a big difference in network reliability between the current largest blockchain network (Bitcoin) and the alternate blockchains.
3. Selfish-mining like behavior can emerge by itself in blockchains.
4. Consensus-breaking changes are fundamentally hard. They are not just engineering problems, but also involve the incentive structures of the parties involved.
5. At the current stage of blockchains, there are not enough computing cycles dedicated to support multiple secure blockchains.

[12]

Security relies on the network's combined computing power. Introducing new features to a blockchain might need a consensus breaking change, which risks splitting the network on a hard fork, thus reducing the amount of computing power each branch then has. New features also risk introducing new bugs to the code, which also limits what gets added. Security and new features are therefore in conflict, because of the risk of hard forks and bugs. [12]

There are also some technical limitations why using a virtualchain is better than just the blockchain. The next chapter explores these limitations.

Blockchain limitations

Blockchains provide cryptographically auditable append only ledgers. They have no central points of trust or failure and it enables a new class of decentralized applications, where users do not need to put their trust in a single party, like with for example DNS root servers or the root certificate authority. However building a distributed DNS and PKI with just a blockchain has some technical limitations:

- **Space**. Individual blockchains records hold data in the order of kilobytes only. [17]
- **Latency**. Speed of creating and updating records is capped by the blockchains write propagation and it is typically in the order of 10-40 minutes. [14]

- **Amount of operations**. Total new operations in each round are limited by the average bandwidth of nodes participating in the network. Currently at around 1400 new operations per block. [6]

- **Bootstrap time**. New nodes need to independently audit the global log from beginning. Time to bootstrap new nodes increases linearly with time.

Using a virtualchain helps to overcome these limitations and to create a system that binds human readable *names* to arbitrary *values*. The underlying blockchain provides consensus on the global state of the naming system and provides an append-only global log for changes. Writes to name-value pairs can only be announced in new blocks. The global log is logically centralized (all nodes see the same state) and organizationally decentralized (no central party control the log). The virtualchain is used to overcome the problems with space, latency, operation count and bootstrap time. [12]

The next section explores the Blockstack architecture to explain how the virtualchain solves these problems.

Blockstack Architecture

Blockstack enables users to register unique human-readable usernames and addresses, and associate public-keys, like PGP, along with additional data. Blockstack does this by using the Bitcoin blockchain and by separating the control and data planes: only minimal metadata (data hashes and state transitions) are kept in the blockchain and external datastores are used for bulk storage. New functionality is introduced to production blockchains without consensus-breaking changes with the use of a logically separate layer, virtualchain. [12]

Some challenges with blockchains:

- **Limits on data storage**. Individual blockchain records can only hold data in the order of kilobytes. Also the full blockchain keeps increasing in size and is currently over 100GB in size.

- **Slow writes**. Transaction processing rate is capped by write propagation. New blocks are announced every 10 minutes and a transaction might take something between a couple of minutes to a few hours to be accepted.

- **Limited bandwidth**. Number of transactions per block is limited by the block size of the blockchain. Block size is usually limited by the

average uplink bandwidth of nodes. For bitcoin this is about 1MB per block (1000 transactions).

- **Endless ledger**. New nodes take a considerable time to boot up, as they need to verify every block from the start.

[12]

Blockstack constructs the naming system as a separate layer on top of the underlying blockchain. The underlying blockchain is used to achieve consensus on the state of the naming system and to bind names to data records. The blockchain is used as a communication channel to announce state changes and provide total ordering for all operations supported by the naming system, like name registrations, updates and transfers. [12]

Blockstack separates the control and data plane. The control plane defines the protocol for registering human-readable names, creating (name,hash) bindings and creating bindings to owned cryptographic keypairs. The control plane is the blockchain and a logically separate layer on top, called a *virtualchain*. The data plane handles the data storage and availability. It consists of zone files for discovering data by hash or URL, and external storage systems for storing the actual data. Name owners sign data values with their private keys. Clients read data values from the data plane and verify that either the data's hash is in the zone file, or that the data includes a signature with the name owner's public key. [12]

The design of Blockstack allows it to run on top of any blockchain and it is able to migrate to a new blockchain if the current one becomes compromised. The security and reliability of Blockstack is directly related to the underlying blockchain. Blockstacks virtualchain concept allows it to create arbitrary *state machines* after processing information from the underlying blockchain. A virtualchain treats transactions from the blockchain as inputs to the state machine and valid inputs trigger state changes. This method allows the introduction of new state machines without consensus-breaking changes to the blockchain. Currently Blockstack introduces a state machine that represents the global state of a naming system. It is possible to construct other state machines using the same method. [12]

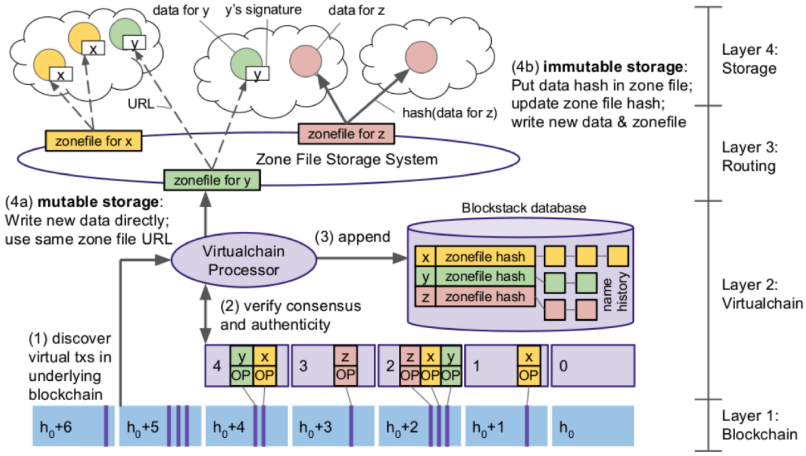

Figure 2.1: Blockstack Architecture [12]

Blockstack architecture consists of four layers that are shown in the figure 2.1. The first layer is the blockchain layer. It stores the sequence of Blockstack operations and provides consensus on the order of the operations. The Blockstack operations are encoded in the transactions of the underlying blockchain. [12]

The second layer is the virtualchain layer. It defines the operations of the Blockstack and new operations can be added without affecting the blockchain layer. Also the rules for accepting or rejecting Blockstack operations are defined in the virtualchain. Accepted operations are processed by the virtualchain to construct a database which reflects the global state of the state machine. Currently Blockstack only has a single state machine - the global naming and storage system. [12]

The third layer is the routing layer. Routing requests (how to discover data) is separated from the actual storage of data. This allows multiple storage services to coexist. Zone files are used to store the routing information, which are identical to DNS zone files in their format. The virtualchain binds names to *hash(zonefile)* and stores this information on the control plane. The actual zone files are stored in the routing layer. Users do not need to trust the routing layer, because the integrity of zone files can by verified by checking the hash of the zone file from the control plane. For storing the zone files, Blockstack nodes form a DHT-like peer network called Atlas. It is white listed in a way that only the zone files which have their hashes published in the control plane, will be added to the Atlas network. [12]

The fourth layer is the storage layer. It stores the actual data values of

name-value pairs. All stored data is signed by the key of the owner of the name. Storing data values outside of the blockchain results in no size limit to data and allows multiple storage backends. Users don't need to trust the storage layer as the integrity of the data can be verified from the control plane. Two different storage modes are allowed: mutable and immutable. In mutable storage the user's zone file contains a URI record that points to the data that includes a signature of the user. Changing the data involves signing the data again, but there is no need to update the zone file and make blockchain transactions. This is thus as fast as changing the data and signing it. Immutable storage is otherwise similar to mutable storage, but a additional TXT record is added to the zone file which contains *hash(data)*. Changing immutable data requires a update to the zone file, which then requires a new transaction in the blockchain. This is a lot slower and should only be used for data that does not change often and for which it is important that the readers always see the most up to date data. [12]

Blockstack summary

Blockstack is a distributed DNS and PKI built on top of Bitcoins blockchain using a virtualchain. It separates functionality in to four different layers: blockchain, virtualchain, routing and storage layers. The virtualchain is build on top the blockchain using the blockchains additional data field called *OP_RETURN*. Virtualchain rules and operations related to name registrations are stored in this field using hashes and operation codes. This linking brings the same trustlessness feature to the virtualchain as is in the underlying blockchain.

Blockstack was first built to use its own blockchain implementation called NameCoin. This turned out be a bad idea, as the NameCoin blockchain had very little computing power, which exposed it to 51% attacks. This is why using the most popular (largest computing power) blockchain is preferred, as Blockstack is only as secure as the underlying blockchain. Currently Blockstack uses the Bitcoin blockchain.

Next a brief look in to the setup behind different parts of the implementation, before the discussing the actual methods and implementation.

Chapter 3

Setup

The test setup consists of four main parts: The Bitcoin's blockchain, Blockstack, the OMI-Node and a chrome extension called OmiViewer as a simple GUI. Bitcoins blockchain will be used by Blockstack, which will be used to register an unique id and to link the OMI-Node address and public key file cryptographically to the id. These features will be used "as is" without any modification. Only development related to these parts will be the OMI-Node address and public key file that will be linked in Blockstack's zone file. The actual development part is done on the OMI-Node's message interface, by adding hashing and signing of objects, and the OmiViewer is developed from scratch to work as a simple proof of concept GUI to find OMI-Nodes linked to Blockstack id's and to verify the data published by OMI-Nodes.

Bitcoin and blockchain

The work is done using production Blockstack, which runs on the current production Bitcoin blockchain network. At the time of writing the current Bitcoin Core version is 0.14.2 [3]. Bitcoin also just activated a major update called the Segregated Witness (SegWit) which among other things improved block capacity and size, and fixed bugs [4]. The Bitcoin Core program is not installed or used directly in this thesis, but Blockstack's default Bitcoin node is trusted and used for buying and using bitcoins for name registration and updates. Even this trust step could be avoided by installing the Bitcoin-core software and using a local copy instead of the default one run by Blockstack's development team.

To buy the Blockstack unique id address, bitcoins are needed. These were bought using the Finnish service Bittiraha.fi [5].

Blockstack

Blockstack will be installed on a Ubuntu 16.04 LTS operating system on a laptop. Blockstack version 0.14.5.1 is installed from Blockstack's github [7]. Specifically the blockstack-core program is installed following the github instructions. Blockstack-core is the main application used to control the linked bitcoin wallet, to buy unique id's and to update the zone file.

OMI-Node

The O-MI-Node developed at Aalto is modified to hash and sign response messages [1]. The O-MI Node is a *Internet of Things* data server that uses *Open Messaging Interface* (OMI) and *Open Data Format* (ODF). It is developed mostly in Java and Scala, and the necessary modifications done by me to the response messages are done in Scala. The O-MI Node will be installed, developed and used locally in the same laptop running Ubuntu 16.04 LTS.

OmiViewer chrome extension

The OmiViewer chrome extensions will be developed on the current version of the Chrome browser, which at the moment of development is 58.0.3029.81 (64-bit). It will also be installed and developed locally on the same laptop running Ubuntu 16.04 LTS.

In the next chapter I will go through the plan on a higher level.

Chapter 4

Methods

The purpose is to develop a proof of concept where the trustless feature of the blockchain is carried over to the data sharing of an OMI-Node. In this chapter I will go through all of the parts from the blockchain to the OMI-Node and explain how the trustless feature is transferred from step to step. I will also describe on a higher level what needs to be developed to achieve this.

Bitcoin blockchain

The trustlessness starts from Bitcoin's blockchain which is explained in more detail in the *Background* chapter. To reiterate, the blockchain is a cryptographically linked chain of blocks which forms a unforgeable public ledger. The blocks contain transaction records on public addresses which hold the actual value (bitcoins) and are controlled by users with their corresponding private keys. Transactions can only be made with the private keys and forgery is practically impossible. The network and the generation of new linked blocks is handled by *miners* who compete on solving a cryptographically hard hash problem. The problem consists of finding a nonce, that when hashed with the previous blocks hash, the next transactions and the nonce, results in a predetermined amount of zeros in the resulting hash.

From the perspective of this thesis, what is needed from the Bitcoin blockchain, is two Bitcoin addresses. One to be used to pay for the Blockstack id in bitcoins and one to be used as the data address to hold Blockstack specific transactions used in the virtualchain. The virtualchain uses a feature called **TX_NULL_DATA or OP_RETURN** in Bitcoins blockchain which allows additional data to be stored in transactions. More detailed description of this can be found in the chapter *Background*. This links the trustless

feature to Blockstack's virtualchain.

Blockstack

Trustlessness feature is brought to Blockstack's DNS by using the additional data field (**TX_NULL_DATA or OP_RETURN**) in Bitcoins blockchain to store virtualchain operations codes and a zonefile hash. In this thesis the Blockstack api will be used to register a unique id and to update the profile of the id to hold a link to a json file with OMI-Node information. This file will be signed with the blockstack api using the data key linked to the registered id. This allows the storing of the file anywhere, as the signature can be used to prove that the file was uploaded by the owner of the id. The file will have the addresses and public keys of the OMI-Nodes, which will be used by the OmiViewer to navigate to the OMI-Node and to verify signed messages.

OmiViewer and OMI-Node

A basic chrome extension will be developed that uses the browser action popup to display a page which allows the user to search the Blockstack for an registered id. The extension will then retrieve the zone file behind the id, parse it for OMI-Node urls and public keys, and then display them for the user. When a url is clicked it takes the user to the OMI-Node. Then when a user makes a search that returns Omi xml data, a verify button can be pressed, which checks the returned xml for a signed hash and verifies that it has been signed with the private key corresponding to the public key that was retrieved from Blockstack. The OMI-Node xml responses in the GUI will be extended to include this signed hash of the objects.

Checking the signature of the signed hash of the OMI-Node message is the last link in the trustless chain. The trustlessness is preserved cryptographically through the whole chain: starting from the Bitcoins blockchain, then through Blockstack DNS to the signed json file with OMI-Node addresses and public keys, and finally ending in the OMI-Node signed hashes of the response messages.

In the next chapter I will go through the implementation in more detail.

Chapter 5

Implementation

In this chapter the actual implementation is presented in more detail.

Bitcoin blockchain

In my implementation I decided to use the default Bitcoin-core server provided by Blockstack developers. This is a additional part that needs to be trusted, but it suffices for the purposes of this thesis. It would however be possible to run the Bitcoin-core server ourselves as well, thus removing the need to trust the server ran by Blockstack's developers. The server address used is *bitcoin.blockstack.com*.

The Bitcoin blockchain is mainly used in the background by the Blockstack program, which is covered in the next chapter.

Blockstack

Blockstack was used to do the following operations: create a bitcoin wallet, register a unique id, update the zonefile and to sign the linked json file used with OmiViewer.

Creating a wallet

After setting up the Blockstack-core node and starting it with the command *blockstack-core –debug start* a wallet is created automatically. The wallet address can be queried with the command *blockstack deposit* which returned the address *3G6RjfGAdTrCm7DKSfevFrmJwiczsq6Wfw*. I used this address to transfer bitcoins to the wallet. The bitcoins were bought using the Finnish service *Bittiraha.fi*. Before starting the registration process the balance of

the wallet should be checked to be sure that the funds have been transferred. This was done with the command *blockstack balance*.

When the funds had been transferred, the registration process could be started.

Registering a unique id in Blockstack

The registration process should be started by first checking the price of the wanted id and if it is available. Availability could be easily checked by doing a standard lookup with the command *blockstack lookup <name>.id*. The price could be checked with the command *blockstack price <name>.id*. I wanted to register a id that I could use in the future for other purposes also, so I decided to register my nickname *jukuli.id*. The price command returned a total estimate of 0.0026488 btc for this id.

After checking the availability and making sure that there is enough funds in the wallet, the registration was done with the command *blockstack register <name>.id*. This starts the registration and should return a success message stating that the registration is pending. The whole registration process has a couple of steps that are done in the background which needs a few blockchain transactions. Because of this the registration process might take a few hours. The progress can be monitored with the command *blockstack info* and when it is complete, the registered id should show up with the command *blockstack names*, which lists all the registered names.

The next step in the process was to update the zone file to have a link to a json file storing the OMI-Node information.

Updating the zone file

After the id had been registered it could be viewed with the command *blockstack whois <name>.id* and *blockstack lookup <name>.id*. The first one returns technical information:

Listing 5.1: Whois: Technical information


```
{
"block_preordered_at": 461982,
"block_renewed_at": 461993,
"expire_block": 567183,
"has_zonefile": true,
"last_transaction_height": 462004,
"last_transaction_id": "
    d2ea0bb984f350bae6c9a7a79f64ac0486243548de1c6080cb122b6536cec32b
```

```
    ",
"owner_address": "37XbxF8QSn2PoMc8vuTng7kdu7JufAhcKW",
"owner_script": "
    a914400900574786463f2665f9c1082465f5dd361da287",
"zonefile_hash": "
    bed6805080598c496ee8b7cf359e75e27e4e9934"
}
```

The second one returns the profile and zone files:

Listing 5.2: Lookup: Profile and Zone files

```
{
"profile": {
"@type": "Person",
"account": [
{
"contentUrl": "https://dl.dropboxusercontent.com/s/
    g7h2zcwhmqsuo1o/omi_nodes.json",
"identifier": "omi_nodes",
"service": "omiviewer"
}
],
"accounts": [],
"timestamp": 1505912271.21517
},
"zonefile": "$ORIGIN jukuli.id\n$TTL 3600\npubkey TXT
    \"pubkey:data:03
    ee2916f2cfb8dfa9c57a8f8fe00ca0f4d5b648d310ab2c4ff2fee4cf07253e95
    \"\n_file URI 10 1 \"file:///home/jrasi/.blockstack/
    storage-disk/mutable/jukuli.id\"\n_https._tcp URI 10
     1 \"https://blockstack.s3.amazonaws.com/jukuli.id
    \"\n_http._tcp URI 10 1 \"http://node.blockstack.org
    :6264/RPC2#jukuli.id\"\n_dht._udp URI 10 1 \"dht+udp
    ://c83ac230b6d85d34d3171b855688f143c44f7f2d\"\n"
}
```

I decided to use the profile file to store the link to the json file, as changing it only needs a mutable update which can be handled inside the Blockstack virtualchain and does not need a costly blockchain transaction. The profile file is a default functionality added to Blockstack to store personal information about the user. By default it is stored in the Atlas network, but other storage providers can be added, like Dropbox, in similar way as I added the

omi_nodes.json file, as can be seen in the next chapter.

The changes seen, in the file displayed before, were done by using the command *blockstack put_profile <name>.id profile.json* with the contents of *profile.json* being what is displayed in 5.2.

The next step is to sign and store the *omi_nodes.json* file linked in the profile file.

Signing and storing the omi_nodes.json file

The file used to store the OMI-Node addresses and public keys is to be signed with a data private key linked with the registered id. This was done using the command *blockstack sign_data <filename>*. Running this to the *omi_nodes.json* file results in:

Listing 5.3: ominodes.json file

```
bsk2.03
    ee2916f2cfb8dfa9c57a8f8fe00ca0f4d5b648d310ab2c4ff2fee4cf07253e95
    .ZIo+hz8X/h3dTDbmVL2N9OS/m5eqrAC+
    pNuTNpjCcLJOPGOQoi2fRVXaSecpNVzjRicAZYQl4kWyMKSNfO/
    pjA==.277:"{\n\t\"omi_nodes\":[\n\t\t{\"url\":\"http
    ://localhost:8080/html/webclient/index.html\",\n\t\t
    \"public_key\":\"25X+V4A6uVqXM/
    CQqKjQKikqvzYDA0Hd27dfXCDH4+s=\"\n\t\t}\n\t],\n\t\"
    signature_url\":\"https://dl.dropboxusercontent.com/
    s/9vdm66sdeqmc6ta/omi_nodes_signature.txt\"\n}\n",
```

Now this is saved to the *omi_nodes.json* file and stored in the url displayed in the profile file: *https://dl.dropboxusercontent.com/s/g7h2zcwhmqsuo1o/omi_nodes.json*. I decided to use Dropbox to store the file, but any file service could be used (or multiple) as signing the file removes the need to trust the underlying service.

After the *omi_nodes.json* file with the OMI-Node links and public keys were linked in the Blockstack profile I could start working on the OmiViewer.

OmiViewer and OMI-Node data signing

The OmiViewer is a simple chrome extension developed by me to handle a couple of functions: to find the desired id from Blockstack DNS, to retrieve the *omi_nodes.json* file and to verify the hashed and signed OMI-Node response data using the retrieved public keys.

Finding a user by id

When a user types an id in the search bar, the extension does a user lookup in the Blockstack network. This can be done using the local api generated by the Blockstack program, which by default can be found running in *localhost:6270*. However, I had some problems with the local api, which halted my progress significantly, so I decided to use the api provided by the Blockstack developers in the url *https://core.blockstack.org/*. This adds another point that needs to be trusted, but which can easily avoided in a full fletched implementation by running the api ourselves.

The OmiViewer does the Blockstack user lookup in the background by making a REST call to the address *https://core.blockstack.org/v1/users/<userid>*. This returns the same profile and zone file information returned by the blockstack program as in listing 5.2.

Parsing the user profile

The profile is parsed to find the link to the *omi_nodes.json* file and then this file as displayed in listing 5.3 is retrieved. The profile is also parsed for *pubkey:data* which is the public data key and can be used to verify the data that is signed and stored externally, like the *omi_nodes.json* file. Next the *omi_nodes.json* file should be verified, but that turned out to be problematic. I was not able to use the same libraries for signature verification as I used later on with the OMI-Node data. After trying multiple different libraries I had to skip this part, because of lack of development time. This adds yet another part to trust in with this proof of concept implementation. I am however quite certain that with more time a suitable library can be found to verify the signature made by the Blockstack program.

After the supposed verification of the *omi_nodes.json* file signature, the file is parsed for OMI-Node urls and public keys, which are then displayed to the user. Clicking a url takes the user to the OMI-Node's web interface and stores the public key for later signature verification.

OMI-Node data hashing and signing

The OMI-Node adds a signature of the hashed objects in the response xml message. Every *<result>* tag has a *<return>* and a *<msg>* tag. The hash and signature is calculated from the *<Objects>* tag that is inside the *<msg>* tag and from everything inside it. The hash and signed hash is added to the *<return>* tag as attributes *objectsHashed* and *hashSigned*.

The used OMI-Node implementation is a modified version of the Aalto OMI-Node [1]. Modifications were done to the *OmiService.scala* class in the *handleRequest* function and in the *OmiConfigExtension.scala* class. The *OmiConfigExtension.scala* was modified to add the loading of the private and public keys from the config file to the config object. The *handleRequest* class uses two new libraries to add the hashing and signing. *Scrypto* [10] library is used for hashing and the *nacl4s* [9] library is used for signing. The <*Objects*> tag and its contents are read as a string, which is then hashed with a SHA-512 hash. This hash is then signed with the private key that is stored in the OMI-Nodes config. The resulting signature and hash are both then encoded with a Base64 encoding and stored to *hashSigned* and *objectsHashed* attributes in the <*return*> tag.

Verifying signature in OmiViewer

After the OMI-Node has returned the response xml, the user can use OmiViewer to verify that the signature is signed with a private key corresponding to the public key retrieved through Blockstack. When the user clicks a *verify* button, OmiViewer scans the DOM tree for the response message. The signature is then opened using the public key listed in the *omi_nodes.json* file. This is done using a javascript library called *tweetnacl*[11]. The <*Objects*> tag is hashed and the resulting hash is compared to the hash retrieved from opening the signature. If these two hashes are the same, the user knows that the data is really signed by a key owned by the same person that was found from Blockstack by the known id.

summary

By using cryptographic methods, the trustless feature is carried over from Bitcoins blockchain all the way through Blockstack DNS and to finally to the actual data shared by the OMI-Node. Computing power behind the Bitcoins blockchain guarantees that there is no need to trust anything else but the algorithms running the system. This feature is then brought to Blockstacks DNS by cleverly using additional data in Bitcoin transactions, allowing a virtualchain to run on top of it. Blockstack DNS is then used to register a id and to store additional information behind that id, including information about OMI-Node addresses and public keys. Finally the OMI-Node public key is used to verify the data signed by the node and proving that it is really published by the owner of the id originally found from Blockstack DNS.

The next chapter evaluates the implementation and discusses some of the problems encountered during development.

Chapter 6

Evaluation

The original goal of the thesis was to build a proof of concept implementation where the Blockstack DNS is used to extend the trustless feature of Bitcoins blockchain to the data shared by OMI-Nodes. The goal was partially achieved and this chapter discusses what was successful, what failed and what were the problems leading to the failure.

Partial success

The proof of concept was partially successful, although it had some parts that could not be implemented as planned. I was able to successfully register a unique id in Blockstack's DNS and add additional information to the id profile with the *omi_nodes.json* file, that allowed me to link the id to the OMI-Nodes. This additional information could also be found with the OmiViewer chrome extension and shown to the user. I was also able to modify the OMI-Node to add a signature of a hash of objects to the response xml. This signature was also successfully verified with OmiViewer.

However there were a few parts that failed, adding the need for additional trust for the system. Biggest one being the failure to verify the *omi_nodes.json* file that was signed by the Blockstack program with the id's private data key. This and other problems are discussed in the next section.

Failures and problems

The difficulties with the thesis can be separated to two categories: planning and implementation.

Planning

The first mistake I made was to underestimate the difficulty and overestimate the amount of background material. The whole field of blockchains is quite new and source materials are scattered and few. Collecting and writing the background information took a lot longer than I first anticipated and this cut time from the actual implementation. Blockchain information is also changing in a fast pace and this made writing the background chapter especially hard. I was constantly facing the dilemma of using more time to rewrite sections or leaving old and incorrect information to the thesis. For example the *Segwit* update to Bitcoin forced me to rewrite parts of the thesis.

These problems could have been at least partially avoided by starting the thesis writing earlier, but it would have also increased the probability for information becoming old during the writing process. I could have also started working on the implementation concurrently while writing the background information. This might have expedited the writing process by making me focus more on information directly related to the implementation.

Implementation

As stated in the previous section, writing the background chapter cut a lot of time from the actual implementation. This in itself would have been enough to cause problems, but there were also problems related to the technologies used.

Blockstack is still in early development and this could be seen as missing documentation and buggy software. I even ended up making a few minor changes to the *blockstack* program to get some of the functionality working correctly. This took away some of my own development time. The *blockstack* program also went through frequent updates, which made changes to the api that forced me to make changes to my own programs as well. In addition, the updates forced me to use the developers own web service for querying Blockstack from OmiViewer, as the local implementation stopped working and I could not figure out why. This introduced the first extra step that needed additional trust, that was not in the original thesis plan.

Finding working cryptographic libraries for the environments used in the planned implementation also turned out be difficult. For example there were multiple libraries that could be used with Scala and Javascript separately, but finding ones that worked together was a lot more difficult. Most libraries had a slightly different way of signing and verifying signatures, that resulted in not being able to verify signatures in Javascript that were signed in Scala. With more development time these differences could have been found and

taken in to account, but instead I decided to find a library that worked in both. I was able to find a library that worked for Scala and Javascript, but I failed to find a library for Javascript that was able to verify the signature done by the *blockstack* program for the *omi_nodes.json* file. This resulted in a second extra step that needed additional trust.

Most of the implementation related problems should be solved when the Blockstack software and cryptographic libraries become more mature. With more development time, I think they could have been mostly solved even now. Currently the technology in the field changes so fast, that it is hard for a single developer to keep up with the changes.

Chapter 7

Conclusions

The proof of concept was only partially successful. However, I think the idea behind is valid and could be implemented with more development time and smarter technology decisions.

Extending the trustless feature of blockchains to other areas, is in my opinion a development that will see a lot of progress in coming years. Integrating the Blockstack DNS directly behind operating systems and browsers will make it more compelling to casual users, allowing more diverse application to be built on top of it.

Blockchains are quite new and the Blockstack distributed DNS is even newer. Still there is a lot of potential and the technology is in a stage that major innovation is already possible. At the moment it just requires a bit more "hands dirty" attitude, as standards and apis are constantly changing and documentation is hard to come by.

Bibliography

[1] Aalto o-mi, github. https://github.com/AaltoAsia/O-MI. Accessed: 2017-10-03.

[2] Bitcoin blockhalf website. http://www.bitcoinblockhalf.com/. Accessed: 2017-10-29.

[3] Bitcoin core, version history. https://bitcoin.org/en/version-history. Accessed: 2017-08-15.

[4] Bitcoin, segregated witness update. https://bitcoincore.org/en/2016/01/26/segwit-benefits/. Accessed: 2017-08-15.

[5] Bittiraha.fi. https://bittiraha.fi/. Accessed: 2017-10-03.

[6] Blockchain.info: Bitcoin transactions per block. https://blockchain.info/charts/n-transactions-per-block. Accessed: 2017-07-10.

[7] Blockstack, github. https://github.com/blockstack. Accessed: 2017-10-03.

[8] Ethereum, blockchain app platform. https://www.ethereum.org/. Accessed: 2017-08-08.

[9] nacl4s crypthographic library, github. https://github.com/emstlk/nacl4s. Accessed: 2017-10-27.

[10] Scrypto cryptographic library, github. https://github.com/input-output-hk/scrypto. Accessed: 2017-10-27.

[11] Tweetnacl javascript library. https://tweetnacl.js.org. Accessed: 2017-10-27.

[12] Ali, M., Nelson, J., Shea, R., and Freedman, M. J. Blockstack: A global naming and storage system secured by blockchains. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016), USENIX Association, pp. 181–194.

[13] Back, A., et al. Hashcash-a denial of service counter-measure, 2002.

[14] Clark, J. B. A. M. J., Edward, A. N. J. A. K., and Felten, W. Research perspectives and challenges for bitcoin and cryptocurrencies.

[15] Diffie, W., and Hellman, M. New directions in cryptography. *IEEE transactions on Information Theory 22*, 6 (1976), 644–654.

[16] Franco, P. *Understanding Bitcoin: Cryptography, Engineering and Economics*. The Wiley Finance Series. Wiley, 2014.

[17] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system, 2008. *URL: http://www. bitcoin. org/bitcoin. pdf* (2012).

[18] Rivest, R. L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2 (1978), 120–126.